

# The Catmull-Clark Surface Subdivision Suite

A few years back I developed a set of macros for the Persistence of Vision Ray Tracer which implemented a Loop subdivision surface scheme, albeit in a limited way. The implementation was hampered by a number of limitations which greatly reduced its usefulness. Then I came out with a set of macros that overcame these limitations to a small degree, by including four-sided faces and implementing support for texture mapping (which had been added to POV-Ray). Now I have made a set of macros which can divide faces of any number of sides, using the Catmull-Clark subdivision scheme.

| Feature                       | Old Surface Subdivision Suite | New Surface Subdivision Suite | Catmull-Clark Surface Subdivision Suite |
|-------------------------------|-------------------------------|-------------------------------|---|
| Border edges                  | Not properly implemented      | Properly implemented          | Properly implemented                    |
| Creases                       | No                            | Yes                           | Yes                                     |
| Individual texturing of faces | No                            | Yes                           | Yes                                     |
| UV mapping                    | No                            | Interpolated                  | Subdivided                              |
| Face types                    | Triangles only                | Triangles and Quadrilaterals  | Any convex face                         |

These macros were developed in conjunction with the [LionSnake modeler](#). Version 1.7 of the modeler uses polygons with any number of sides, and the files which are exported for use by POV-Ray depend on this macro suite to build the subdivided mesh.

## Notable differences:

The NSSS treats the corners of patches just like any other border vertex, which causes them to be rounded off. The Catmull-Clark scheme specifies that border vertices with no interior edge be treated as infinitely sharp corners. This causes patches to behave in the same way as bezier and NURB patches, which makes things easier for people who are used to those objects.

At the moment support for variable-sharpness edges differs slightly from the Catmull-Clark specification. As things stand, child edges inherit the sharpness of their parents (decreased by one). This is within the C-C specs, except that at a crease vertex the child edges are supposed to get their sharpness from a weighted sum of the two creases meeting there. This only matters when the edges meeting at a crease vertex have different sharpness levels.

## The Download

is right here:

[c2s3.zip](#)  
(13,637 bytes)

## Using the file:

Extract the file `c2s3.zip` into the directory where you keep your POV files. At some point in your scene code (prior to invoking any of the macros) insert this line of code:

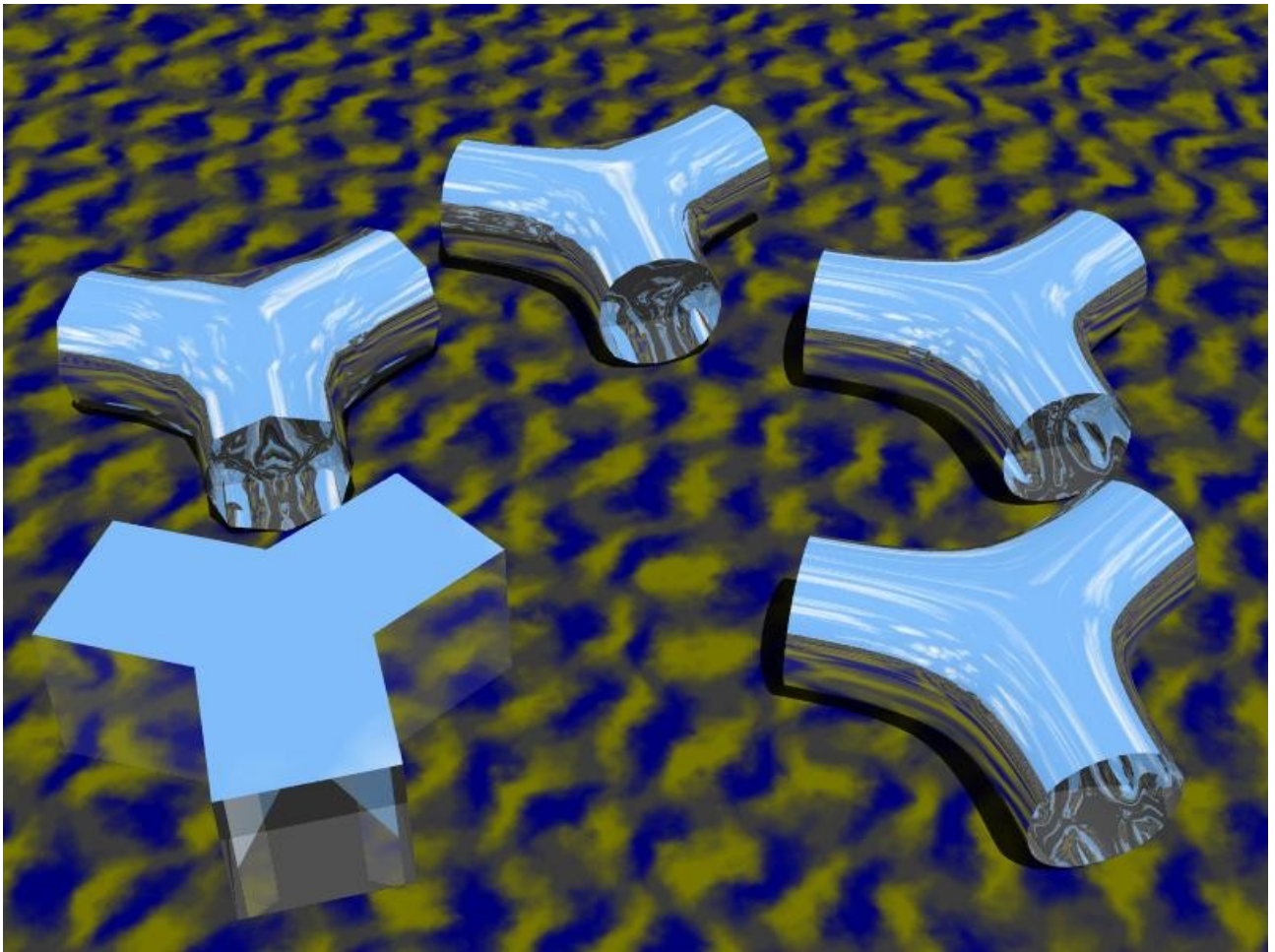
```
#include "c2s3.inc"
```

This file will include all of the other files.

## An example

The following image was created using POV-Ray and the macros. The lower-left object is the original mesh, and clockwise from there are the first, second, third, and fourth refinements of the mesh. You will note that there is no noticeable difference between the third and fourth refinements. The third refinement is enough for most purposes, and going beyond

the fourth refinement is rarely worth the trouble.



## Naming Conventions

The CCSSS makes use of several arrays, variables, and macros, the names of which are written into the code. The labels for the arrays, variables, and macros all begin with `c2s3`. Avoid using labels that start with that, and everything should be fine.

## The Macros

The CCSSS is designed around these new macros. They operate slightly differently from the older subdivision surface macros, so in addition to editing the function calls you'll have to make some other changes as well.

### `c2s3_ImportBasicMesh(Points,Faces)`

This is the macro for importing mesh data into the CCSSS. **Points[]** contains the vertices of the mesh, and **Faces[]** contains a series of face records. Each record consists of a number  $n$  followed by  $n$  zero-based indices into **Points[]**, describing the vertices used by the face.

If any record contains more than or fewer than the specified number of vertex indices, the behavior of this macro is undefined; parsing may stop in this macro with a fatal error, or one of the other macros may stop with a fatal error, or parsing may continue with a corrupted mesh as a result. You have been warned.

This macro copies all of the data in the arrays, so after returning from the macro you can **#undef** (or otherwise modify) the arrays **Points** and **Faces** without affecting the mesh you've imported.

Don't worry about the direction of the vertex windings. The CCSSS will set all of them to the handedness of your scene prior to the first subdivision[2], or if you calculate the normals without first subdividing.

This macro deletes any CCSSS data from earlier parsing in the scene.

### `c2s3_AddTextures(Textures)`

This macro enables you to specify a separate texture for each of the faces in the basic mesh. **Textures** is an array of zero-based indices into an array of textures you will supply later; the order of the entries in **Textures** matches the order of the **Faces** array supplied to the macro **c2s3\_ImportBasicMesh()**. A value of **-1** specifies that no individual texturing is applied for that face.

You don't need an entry for each face you've defined. If you import the mesh such that **Faces** has all of the individually-textured faces at the beginning of the array, and make **Textures** large enough to cover only those faces, the rest will retain the default value of **-1**.

When faces are subdivided, each child inherits its parent's texture. Please note that the CCSSS does not support texture interpolation.

This macro copies all of the data in the array you supply, so after returning from the macro you can **#undef** (or otherwise modify) the array **Textures** without affecting what you've imported.

### c2s3\_AddSharpData(Sharps)

Use this macro to define internal edges (edges with two faces bordering on them) as sharp edges, so that the smoothing calculations are not done for that edge. **Sharps[i][0]** is the index of the starting vertex of the edge to be sharpened, **Sharps[i][1]** is the index of the ending vertex of the edge to be sharpened, and **Sharps[i][2]** is the number of subdivision levels for which smoothing calculations are not done. Fractional values specify interpolation between two integral levels of sharpness. Negative values represent infinite sharpness, but they are converted to 256 internally[1].

Border edges are always treated as infinitely sharp, so there is no need to include them in this array. Likewise, there is no need to specify the sharpness of smooth edges, because smoothness is the default setting.

This macro copies all of the data into its own data array, so after returning from the macro you can **#undef** (or otherwise modify) the array **Sharps[1]** without affecting what you've imported.

### c2s3\_AddTextureMapping(MapPoints,Records)

This macro adds the uv-mapping. **MapPoints[]** is an array of **<u,v>** vectors specifying the texture mapping points used in the mesh. **Records[]** is an array of texture mapping records, with one record for each face in the mesh. The first entry in each record is the number of vertices in the face, and the remaining values are the indices into **MapPoints[]**, specifying the texture mapping point used at each vertex in the face. If you specify **zero** for the number of vertices in a face, then no uv-mapping is applied for that face, and there should be no un-mapping indices in that record.

Note that the first texture map for a given face will be matched to the first vertex for that face, as defined using **ImportBasicMesh()**. If the vertices and texture maps for a given face are specified in a different order, the results will be unpredictable.

If all of the faces with texture mapping are grouped at the start of the imported mesh (using **c2s3\_ImportBasicMesh()**), then the array **Records[]** only needs enough entries for those faces; the remaining faces will receive default texture mapping values.

This macro copies all of the data in the arrays, so after returning from the macro you can **#undef** (or otherwise modify) **MapPoints[]** and **Records[]** without affecting what you've imported.

### c2s3\_Subdivide()

This macro performs one level of subdivision. Each edge is split in two, a new point is added in the center of each face, and then each face is turned into a set of smaller quadrilaterals. The new vertices are calculated so that repeated subdivision results in a smoother mesh. The new mesh is roughly four times larger in memory, so don't go hog-wild. Subdivision beyond the fourth iteration is highly expensive memory-wise and yields little perceptible improvement in object quality.

### c2s3\_CalcNormals()

This macro turns on smoothing of the normals (except at edges defined as sharp) and calculates the normals required at each vertex.

Every call to **c2s3\_Subdivide()** destroys the data created by this macro, so don't bother to calculate the normals until you are done subdividing.

### c2s3\_BuildMesh()

This macro builds a **mesh**{ } object and places it in the scene. The faces are smoothed if you called the **c2s3\_CalcNormals()** macro, otherwise they will be flat (and will show the triangulation of each face with more than three sides). The polygons are not textured by this macro. You can put this call within an **object**{ } statement and apply texturing and transformations to it.

This macro will apply whatever uv mapping was specified if **c2s3\_AddTextureMapping()** has been called. Remember that the **uv\_mapping** keyword must appear in the definition of the texture that you supply.

### **c2s3\_BuildTexturedMesh(Textures)**

This macro builds a **mesh**{ } object and places it in the scene. The faces are smoothed if you called the macro **c2s3\_CalcNormals()** since the most recent subdivision (or mesh import), but the textures in **Textures[]** are applied to every face (or child of a face) that received a non-negative index from a call to **c2s3\_AddTextures()**; the remaining faces receive the default texture. You can put this call within an **object**{ } statement and apply transformations and/or supply a different default texturing.

This macro will apply whatever uv mapping was specified if **c2s3\_AddTextureMapping()** has been called. Remember that the **uv\_mapping** keyword must appear in the texture definitions.

### **c2s3\_JitterVertices(Seed,Amount)**

This macro randomly perturbs the vertices of the array. **Seed** is the return value from a **seed()** function call, and **Amount** is the amount of perturbing to be done. **Amount** can be a vector value, which is multiplied to each component of the jitter (calculated separately).

The jitter value is calculated by adding up twelve random values from zero to one, and then subtracting six. This yields a bell-shaped curve of random values, centered around zero, with a standard deviation of one.

### **c2s3\_BuildFrame(Radius,Index,Color)**

This macro, which I wrote for debugging purposes, creates a set of cylinders for each edge of the mesh. **Radius** specifies the radius of the cylinders. **Index** specifies the zero-indexed edge that is to be colored using the color specified by **Color**.

### **c2s3\_Porcupine(Index,Color)**

This macro, which I wrote for debugging purposes, creates a set of cones with their bases in the center of each face and pointing along each face's normal. The length and radius of each cone is roughly derived from the area of the face. **Index** specifies the zero-indexed cone that is to be colored using the color specified by **Color**.

### **c2s3\_DumpData()**

This macro dumps some of the mesh data to the debug stream.

### **c2s3\_NukeData()**

This deletes all of the data created by the macros. Use this to save memory during parsing.

To reduce memory space contentions, I recommend using the CCSSS as early as possible in scene parsing, and where possible handle the simplest meshes first.

## **Recommended use:**

The order in which macros are called is fairly important. I have tried to idiot-proof things, and since POV-Ray, by its nature, discourages idiots, there should be few problems, but here's the advice:

1. **c2s3\_ImportBasicMesh()** should always be the first macro called for any mesh.
2. **c2s3\_AddTextures()**, **c2s3\_AddSharpData**, and **c2s3\_AddTextureMapping()**, in any order, to add the extra data.
3. **c2s3\_Subdivide()** enough times to achieve the desired subdivision level.
4. **c2s3\_CalcNormals(b)** if you want smoothing.
5. **c2s3\_BuildMesh()** or **c2s3\_BuildTexturedMesh()** to generate the mesh object.
6. **c2s3\_NukeData()** to remove the data from memory when it is no longer needed.

## Other Macros and the Internal Data

Although I included this information in previous versions of this page, I'm keeping them out this time, so that people won't write scene code that may break when I make internal changes. I am considering making a significant internal change to the data structures (for efficiency reasons).

## Version History

### 22 Mar 2008

The macro that allegedly fixed the face winding problem was only pretending to work; in fact, it was messing up the windings of meshes that already had the windings right. That macro was fired, and I hired a new one to replace it.

### 24 Feb 2008

The last bug fix apparently fixed nothing. This one appears to work.

### 23 Feb 2008

The macro that ensured the correct winding of the faces, didn't. Fixed.

### 16 Feb 2008

Original version released to worldwide acclaim.

## Bug Reports

You can e-mail me at **hotmail.com** using the username **evilsnack**.

---

[1] 256 levels of subdivision is far and away enough for any application; if you had a polygon that was twenty billion light-years across (roughly the size of the known universe), and subdivided it 128 times, the resulting child polygons would be around one-millionth the size of a molecule of water. Even subdividing a dozen times will generally result in all of your memory being consumed if you start with more than a small handful of faces (there would be at least 12 million child polygons for each original, roughly the same number of vertices, with a minimum of 24 bytes for each vertex and 16 for each face; 480 megabytes), so a maximum of 256 is certainly overkill.

[2] The order of the vertices only affects the calculation of normals. However, the faces produced by subdivision inherit their winding from the parent face, so arranging them before the first subdivision has only a minimum cost. For that reason, the subdivision macro checks to see if the faces have been oriented, and if not, the macro to orient them is called.

---

**[Back to John's Freeloading Home Page](#)**

Hosted by [www.Geocities.ws](http://www.Geocities.ws)